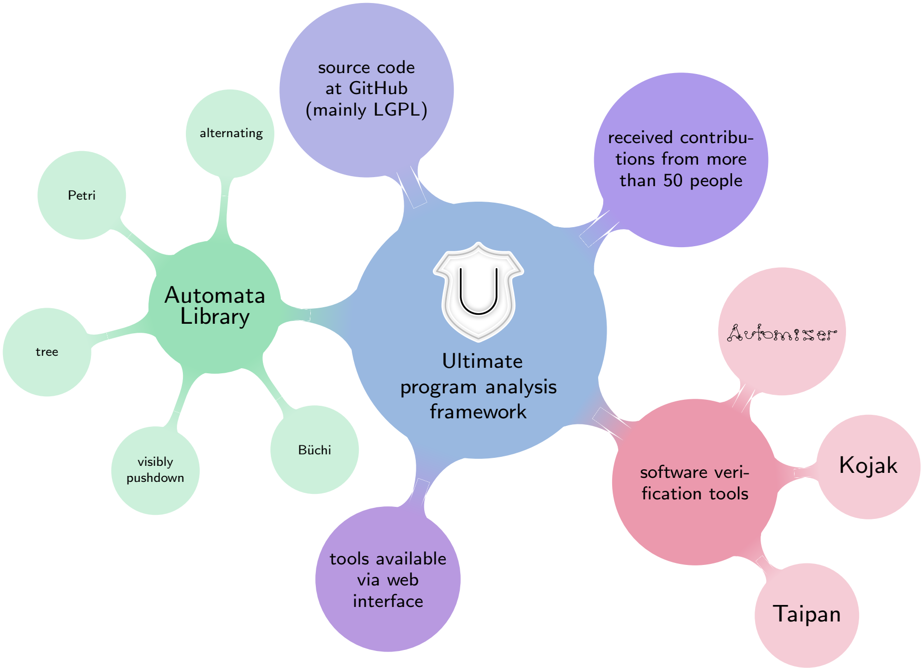
Traces, Interpolants, and Automata: Ultimate Automizer's Approach to Software Verification

Matthias Heizmann

University of Freiburg

EPIT 2018, Aussois, 2018-05-07


source code at GitHub (mainly LGPL)
received contributions from more than 50 people
alternating
Petri
Automata Library
tree
visibly pushdown
Büchi
Ultimate program analysis framework
Automizer
software verification tools
Kojak
Taipan
tools available via web interface

ULTIMATE

Automizer

automata-based software verification

for **non-reachability**, **memory safety**, **termination**, **overflows**

ULTIMATE

Automizer

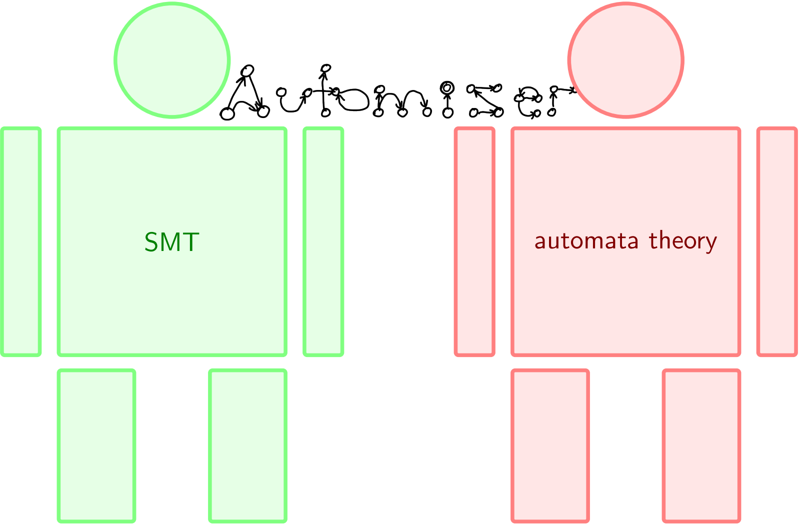
automata-based software verification

for **non-reachability**, **memory safety**, **termination**, **overflows**

Achievements at SV-COMP (`https://sv-comp.sosy-lab.org`)

- 2015: Silver Overall
- 2016: Gold Overall, Gold Falsification Overall
- 2017: Gold Overall, Gold Termination
- 2018: Silver Overall, Gold Termination, Gold NoOverflow

Reason for success? Standing on the sholders of giants!



Outline

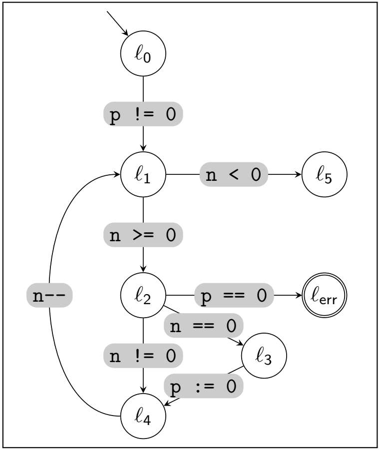
- Software verification:

  Example, Hoare triples, Floyd-Hoare annotation

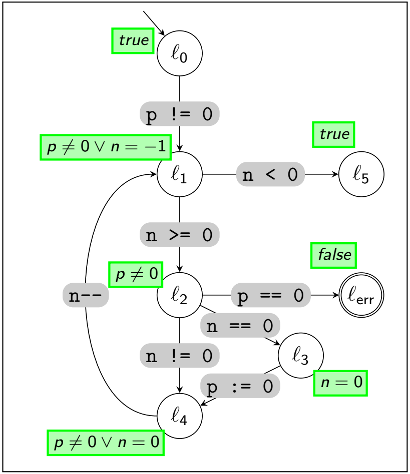
- Trace abstraction: new paradigm
- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code
- Trace abstraction: algorithm
- Termination analysis

```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

pseudocode



control flow graph



control flow graph

Definition:

$\{\boxed{\varphi}\}$ `st` $\{\boxed{\varphi'}\}$ is valid Hoare triple
iff
if program is in state that satisfies φ
and program executes *st*
then program is in a state that satisfies φ'

Example:

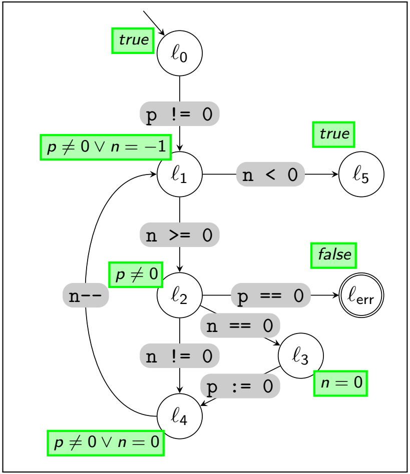
$\{\boxed{p \neq 0 \vee n = -1}\}$ `n>=0` $\{\boxed{p \neq 0}\}$

is a valid Hoare triple

Example:

$\{\boxed{n \neq 0}\}$ `n--` $\{\boxed{n \neq 0}\}$

is not a valid Hoare triple



control flow graph

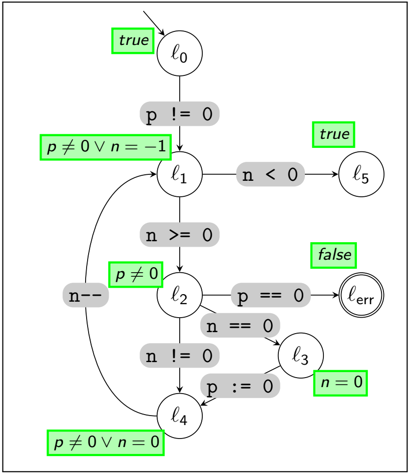
Example:

$$\{\boxed{n \neq 0}\}\ \texttt{n--}\ \{\boxed{n \neq 0}\}$$

is not a valid Hoare triple

SMT script

```
(declare-fun n () Int)
(declare-fun n' () Int)
(assert (not (= n 0)))
(assert (= n' (- n 1)))
(assert (not (not (= n' 0))))
(check-sat)
```



control flow graph

Definition:

A Floyd-Hoare annotation is a mapping that assigns each location ℓ_i a formula φ_i such that there is an edge

$$\boxed{\varphi_i}\ \ell_i \xrightarrow{\texttt{stmt}} \ell_j\ \boxed{\varphi_j}$$

only if the Hoare triple

$$\{\boxed{\varphi}\}\ \texttt{stmt}\ \{\boxed{\varphi'}\}$$

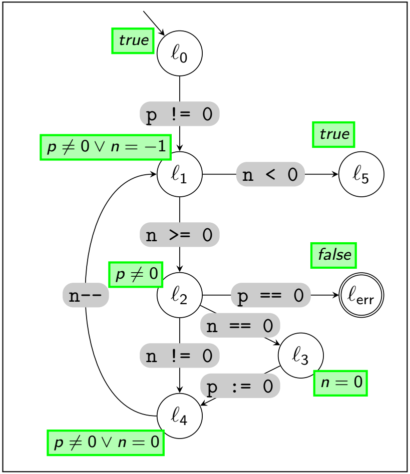
is valid.

Theorem:

Given a program $\mathcal{P}$, if there is a Floyd-Hoare annotation such that

- every initial location is labeled with *true* and
- every error location is labeled with *false*

then $\mathcal{P}$ is correct.



control flow graph

Outline

- Software verification:

  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm

- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code

- Trace abstraction: algorithm

- Termination analysis

New View on Programs

"A program defines a language over the alphabet of statements."

New View on Programs

"A program defines a language over the alphabet of statements."

- Set of statements: alphabet of formal language
  e.g., $\Sigma = \{$ `p != 0`, `n >= 0`, `n == 0`, `p := 0`, `n != 0`, `p == 0`, `n--`, `n < 0`, $\}$

New View on Programs

"A program defines a language over the alphabet of statements."

- Set of statements: alphabet of formal language
  e.g., $\Sigma = \{$ `p != 0`, `n >= 0`, `n == 0`, `p := 0`, `n != 0`, `p == 0`, `n--`, `n < 0`, $\}$

- Control flow graph: automaton over the alphabet of statements
- Error location: accepting state of this automaton

New View on Programs

"A program defines a language over the alphabet of statements."

- Set of statements: alphabet of formal language
  e.g., $\Sigma = \{$`p != 0`, `n >= 0`, `n == 0`, `p := 0`, `n != 0`,
  `p == 0`, `n--`, `n < 0`, $\}$

- Control flow graph: automaton over the alphabet of statements
- Error location: accepting state of this automaton

- Error trace of program: word accepted by this automaton

ℓ_0: `assume p != 0;`

ℓ_1: `while(n >= 0)`

```
{
```

ℓ_2: `    assert p != 0;`

```
    if(n == 0)
    {
```

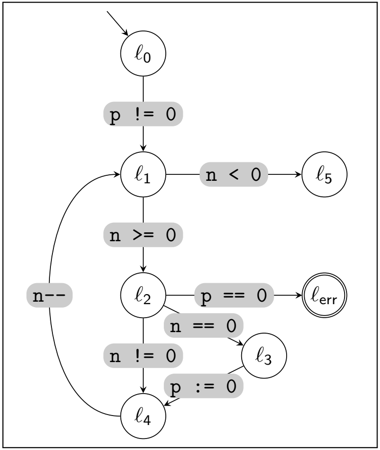
ℓ_3: `        p := 0;`

```
    }
```

ℓ_4: `    n--;`

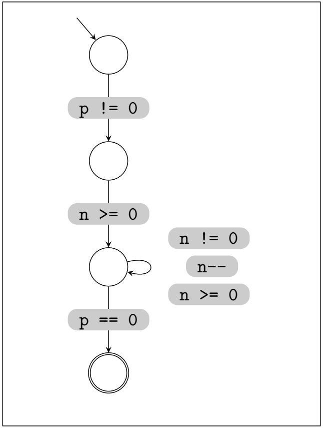
```
}
```

pseudocode



control flow graph

Example



Some program together with a specification in our formalism

Trace Abstraction

Our paradigm:

Computer programs are collections of statements, the definition of a program consists of two parts:

1. meaning of the statements
2. the way how the statements are arranged

Trace Abstraction

Our paradigm:

Computer programs are collections of statements,
the definition of a program consists of two parts:

		defined by
1.	meaning of the statements	semantics of programming language
2.	the way how the statements are arranged	control flow graph (resp. program code)

Trace Abstraction

Our paradigm:

Computer programs are collections of statements,
the definition of a program consists of two parts:

		defined by	our formalism
1.	meaning of the statements	semantics of programming language	SMT
2.	the way how the statements are arranged	control flow graph (resp. program code)	automata theory

Outline

- Software verification:

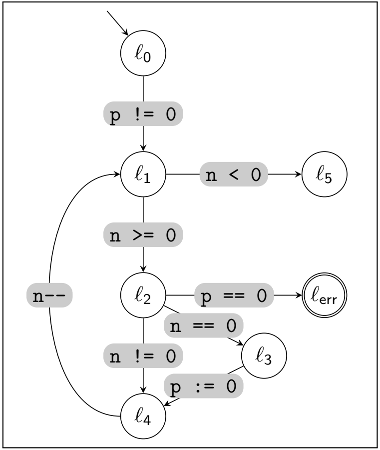
  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm
- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code
- Trace abstraction: algorithm
- Termination analysis

Trace Abstraction: Example

```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

pseudocode

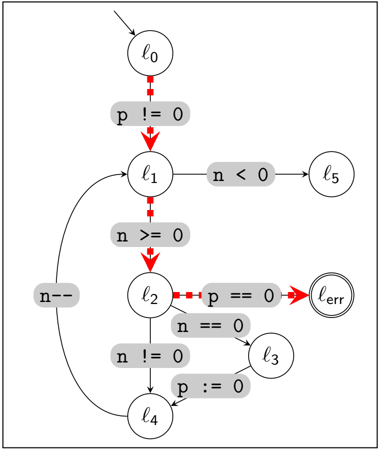


control flow graph

Trace Abstraction: Example

```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

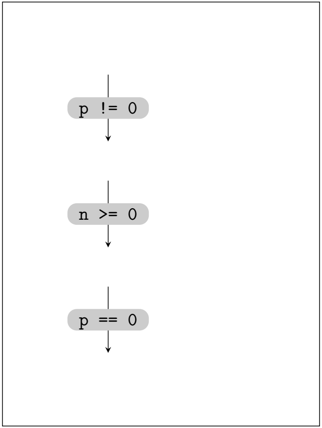
pseudocode



control flow graph

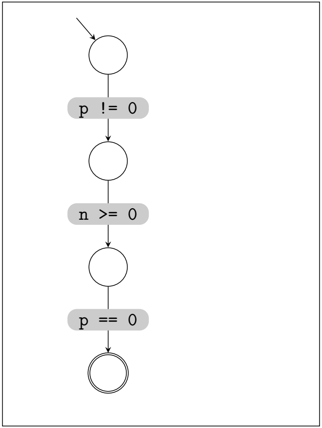
Trace Abstraction: Example

1. take trace π_1



Trace Abstraction: Example

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$




```
1:  assume p != 0;
2:  assume n >= 0;
3:  assert p != 0;
```

pseudocode of $\mathcal{P}_1$

Outline

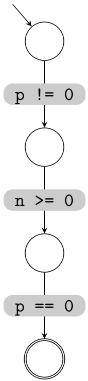
- Software verification:

  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm
- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code
- Trace abstraction: algorithm
- Termination analysis

Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness

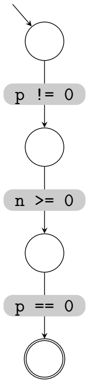


Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness

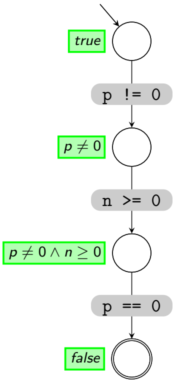
SMT script

```
(declare-fun x () Int)
(declare-fun n () Int)
(assert (not (= p 0)))
(assert (>= n 0))
(assert (= p 0))
(check-sat)
```



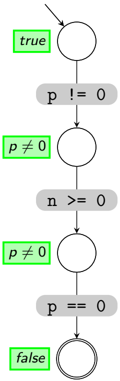
Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness
- Step 2: Construct proof
  - Naive approach: symbolic execution



Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness
- Step 2: Construct proof
  - Naive approach: symbolic execution
  - Alternatives:
    - symbolic execution + unsat cores
    - Craig interpolation

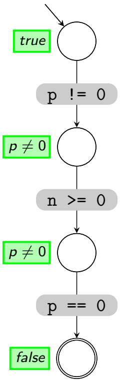


Excursus: Correctness proofs for straightline code

- Step 1: Analyze correctness
- Step 2: Construct proof
  - Naive approach: symbolic execution
  - Alternatives:
    - symbolic execution + unsat cores
    - Craig interpolation

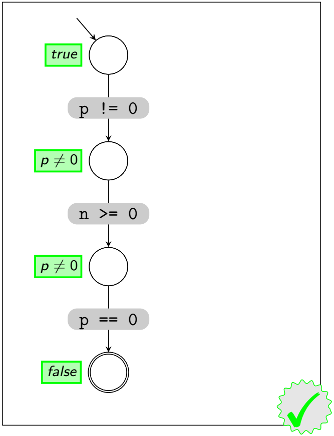
SMT script

```
(declare-fun x () Int)
(declare-fun n () Int)
(assert (!  (not (= p 0)) :named stmt1))
(assert (!  (>= n 0) :named stmt2))
(assert (!  (= p 0) :named stmt3))
(check-sat)
(get-interpolants stmt1 stmt2 stmt3)
```



Trace Abstraction: Example

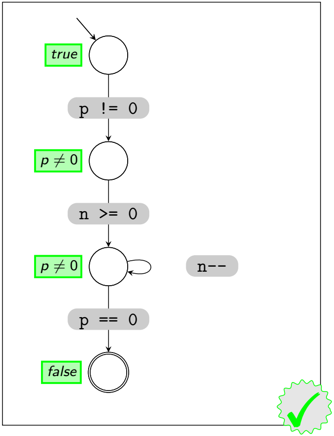
1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$



Trace Abstraction: Example

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

$\{\boxed{p \neq 0}\}$ `n--` $\{\boxed{p \neq 0}\}$ is valid Hoare triple

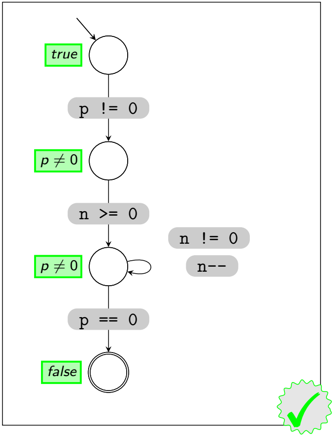


Trace Abstraction: Example

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

$\{\boxed{p \neq 0}\}$ `n--` $\{\boxed{p \neq 0}\}$ is valid Hoare triple

$\{\boxed{p \neq 0}\}$ `n != 0` $\{\boxed{p \neq 0}\}$ is valid Hoare triple



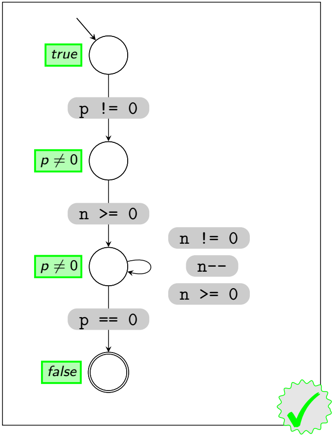
Trace Abstraction: Example

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

$\{\boxed{p \neq 0}\}$ `n--` $\{\boxed{p \neq 0}\}$ is valid Hoare triple

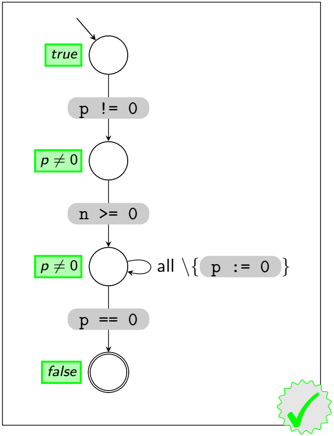
$\{\boxed{p \neq 0}\}$ `n != 0` $\{\boxed{p \neq 0}\}$ is valid Hoare triple

$\{\boxed{p \neq 0}\}$ `n >= 0` $\{\boxed{p \neq 0}\}$ is valid Hoare triple



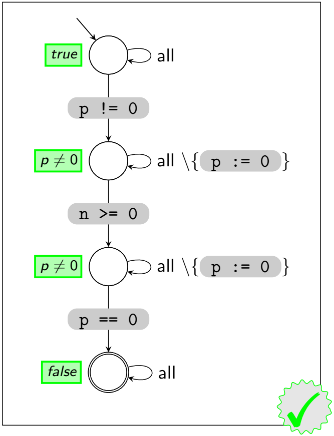
Trace Abstraction: Example

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions



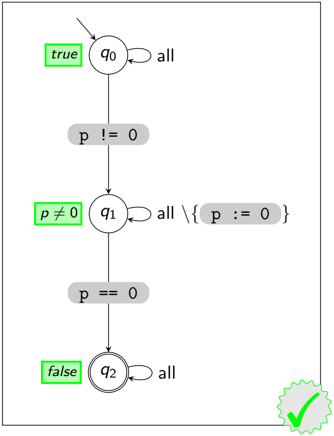
Trace Abstraction: Example

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

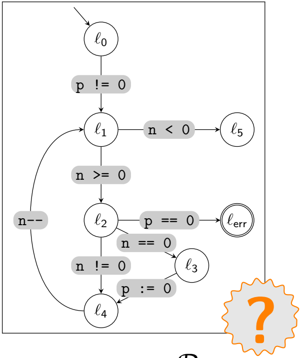


Trace Abstraction: Example

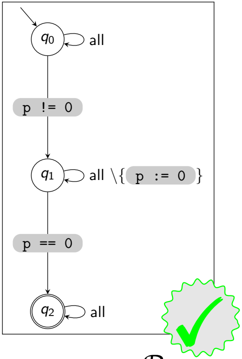
1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions
   - merge locations



Trace Abstraction: Example

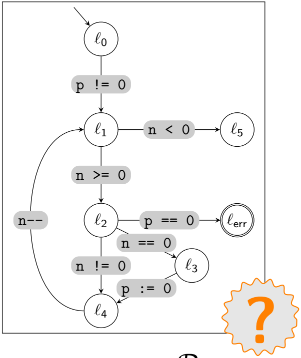


program $\mathcal{P}$

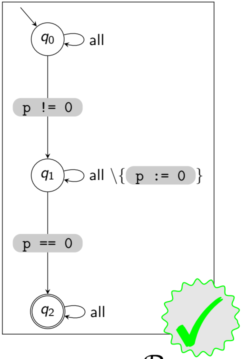


program $\mathcal{P}_1$

Trace Abstraction: Example

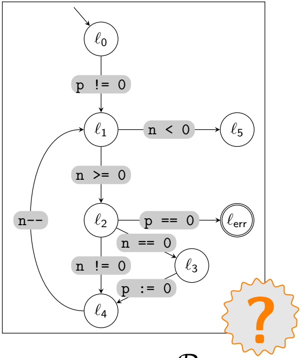


program $\mathcal{P}$

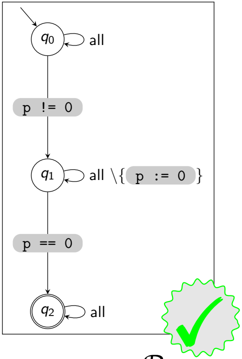


program $\mathcal{P}_1$

Trace Abstraction: Example

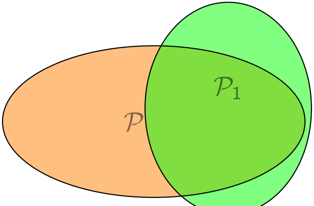


program $\mathcal{P}$

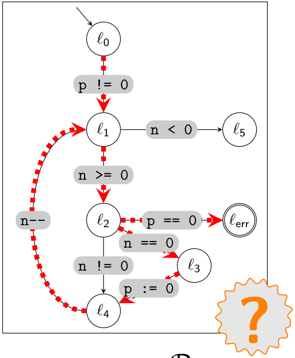


program $\mathcal{P}_1$

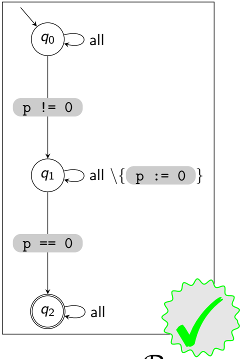
Consider only traces in set theoretic difference $\mathcal{P} \backslash \mathcal{P}_1$.



Trace Abstraction: Example

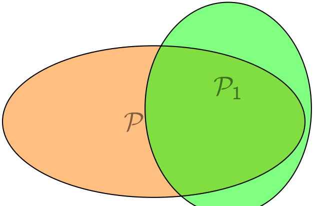


program $\mathcal{P}$



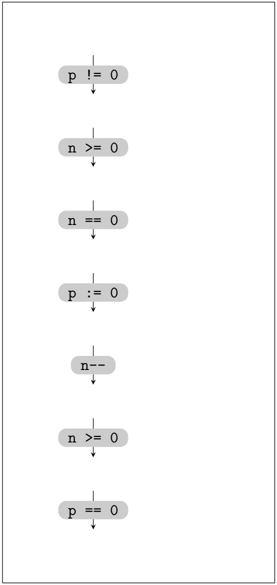
program $\mathcal{P}_1$

Consider only traces in set theoretic difference $\mathcal{P}\setminus\mathcal{P}_1$.



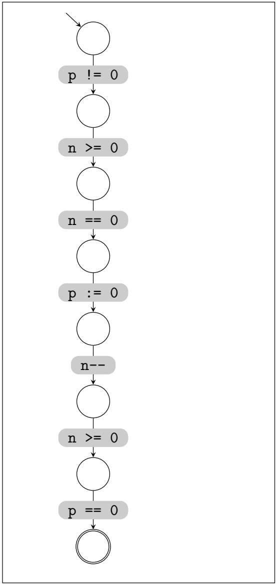
Trace Abstraction: Example

1. take trace π_2



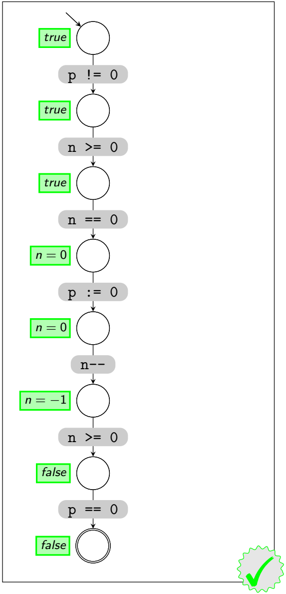
Trace Abstraction: Example

1. take trace π_2
2. consider trace as program $\mathcal{P}_2$



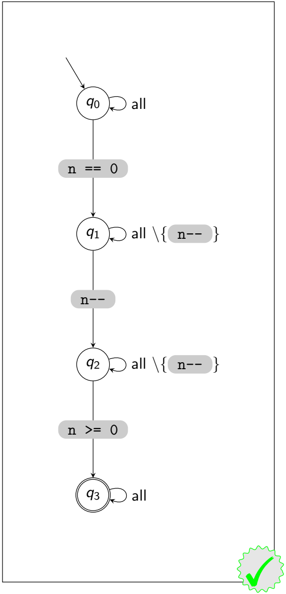
Trace Abstraction: Example

1. take trace π_2
2. consider trace as program $\mathcal{P}_2$
3. analyze correctness or $\mathcal{P}_2$

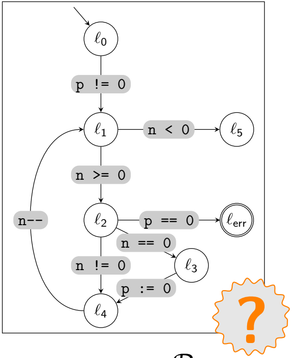


Trace Abstraction: Example

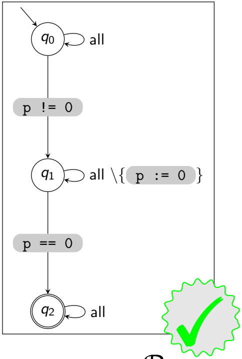
1. take trace π_2
2. consider trace as program $\mathcal{P}_2$
3. analyze correctness or $\mathcal{P}_2$
4. generalize program $\mathcal{P}_2$
   - add transitions
   - merge locations



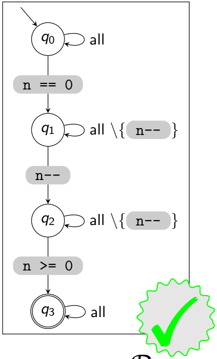
Trace Abstraction: Example



program $\mathcal{P}$

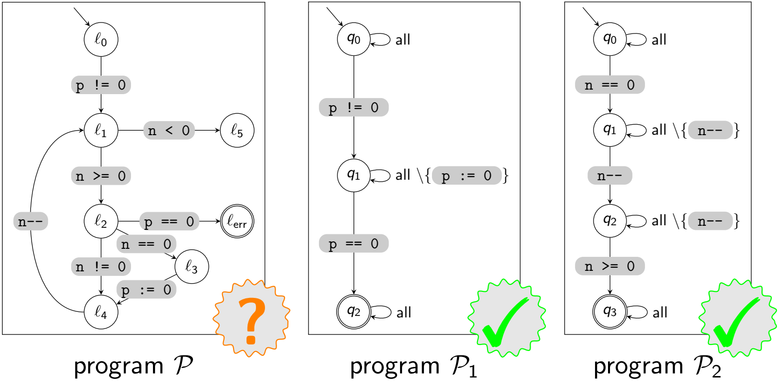


program $\mathcal{P}_1$



program $\mathcal{P}_2$

Trace Abstraction: Example



$$\mathcal{P} \quad \subseteq \quad \mathcal{P}_1 \cup \mathcal{P}_2$$

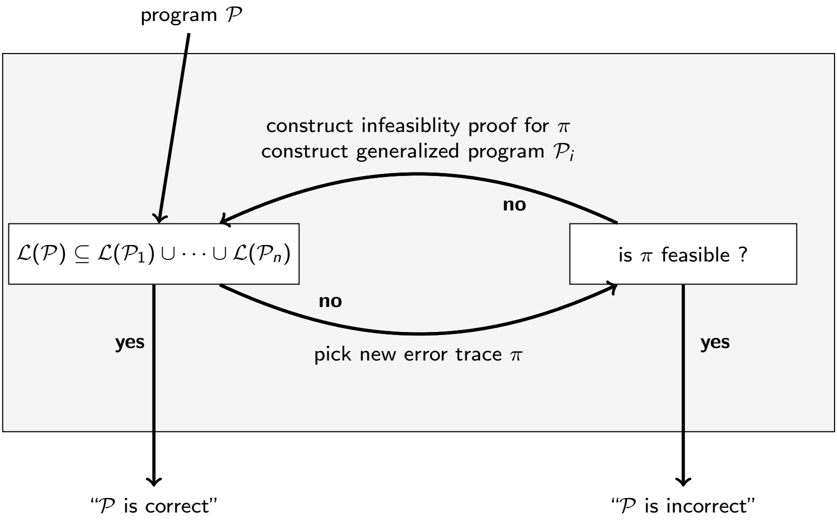
Outline

- Software verification:

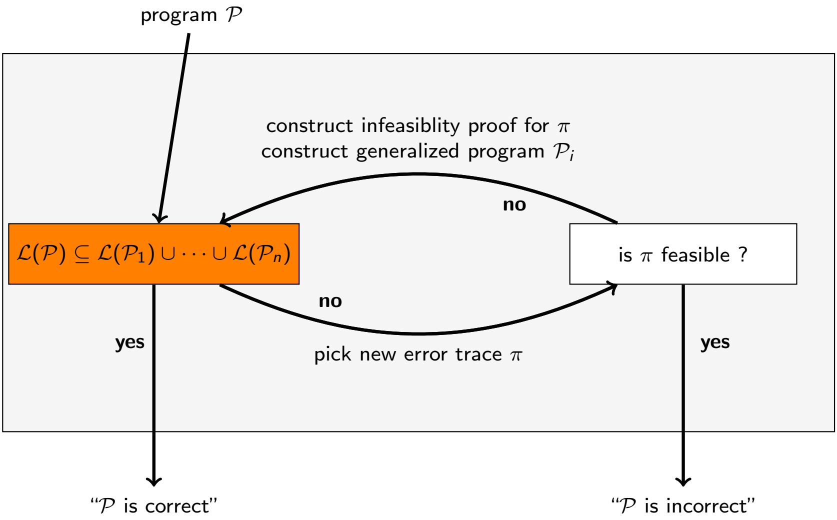
  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm
- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code
- Trace abstraction: algorithm
- Termination analysis

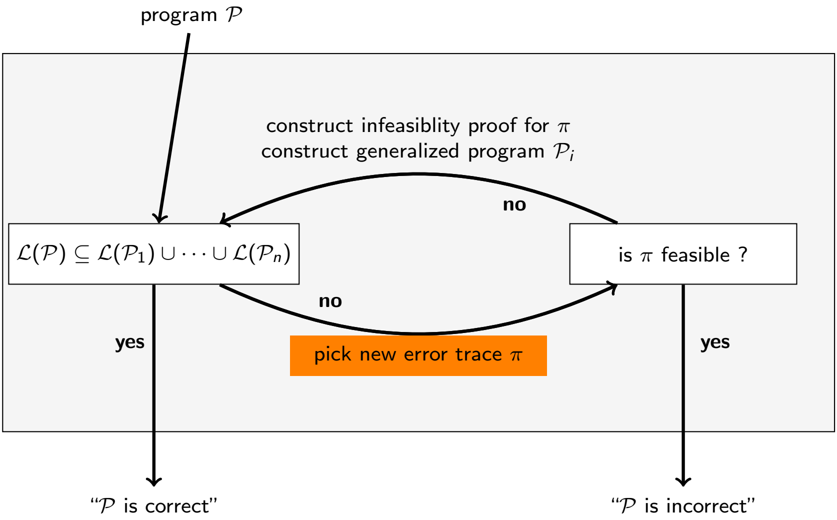
Trace Abstraction: Verification Algorithm



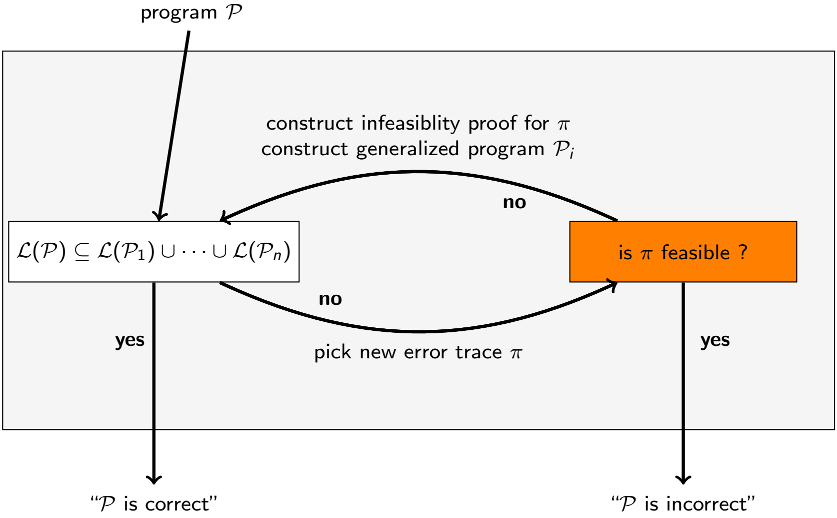
Trace Abstraction: Verification Algorithm



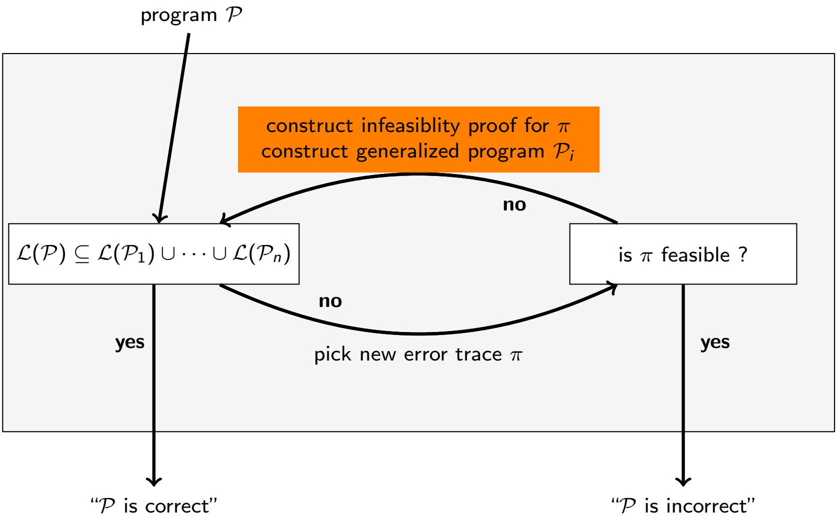
Trace Abstraction: Verification Algorithm



Trace Abstraction: Verification Algorithm



Trace Abstraction: Verification Algorithm

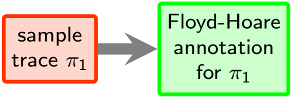


A classical approach to software model checking:

$$\boxed{\text{program } \mathcal{P}} \longrightarrow \boxed{\text{Floyd-Hoare annotation for } \mathcal{P}}$$

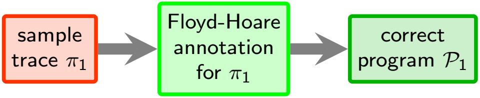
A classical approach to software model checking:

Our approach to software model checking:



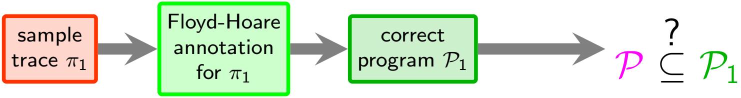
A classical approach to software model checking:

Our approach to software model checking:



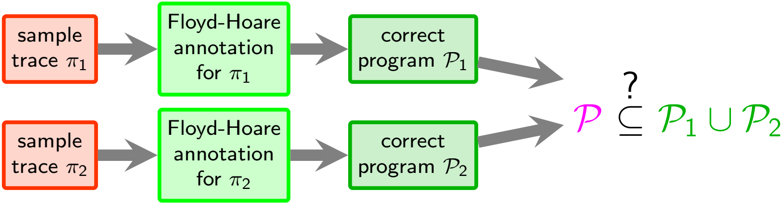
A classical approach to software model checking:

Our approach to software model checking:



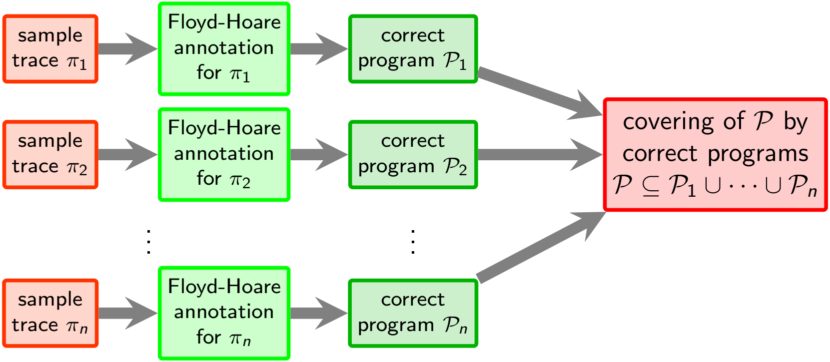
A classical approach to software model checking:

Our approach to software model checking:



A classical approach to software model checking:

Our approach to software model checking:



Extensions of Trace Abstraction Approach

Interprocedural programs

H., Hoenicke, Podelski **Nested Interpolants** (POPL 2010)

Cassez, Müller, Burnett **Summary-Based Inter-Procedural Analysis via Modular Trace Refinement** (FSTTCS 2014)

Concurrent systems

Farzan, Kincaid, Podelski **Concurrent systems** (POPL 2014)

Cassez, Ziegler **Verification of Concurrent Programs Using Trace Abstraction Refinement** (LPAR 2015)

Timed systems

Wang, Sipma **Trace Abstraction Refinement for Timed Automata** (ATVA 2014)

Cassez, Jensen, Larsen **Refinement of Trace Abstraction for Real-Time Programs** (RP 2017)

Solving horn clauses / Tree automata

Kafle, Gallagher **Tree Automata-Based Refinement with Application to Horn Clause Verification.** (VMCAI 2015)

Wang, Jiao **Trace abstraction refinement for solving Horn clauses** (The Computer Journal 2016)

Outline

- Software verification:

  Example, Hoare triples, Floyd-Hoare annotation

- Trace abstraction: new paradigm
- Trace abstraction: example
  - Excursus: Correctness proofs for straightline code
- Trace abstraction: algorithm
- Termination analysis

Termination Analysis

Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

  Solution: consider infinite traces, use ω-words and Büchi automata

Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

  Solution: consider infinite traces, use ω-words and Büchi automata

- Challenge 2: An infinite trace may not have any execution although each finite prefix has an execution.

  E.g., (`x > 0` `x--`)$^\omega$

```
while (x > 0) {
    x--;
}
```

Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

  Solution: consider infinite traces, use ω-words and Büchi automata

- Challenge 2: An infinite trace may not have any execution although each finite prefix has an execution.

  E.g., (`x > 0` `x--`)$^\omega$

```
while (x > 0) {
      x--;
}
```

  Solution: ranking functions (here: f(x)=x)

Ranking Function (for a Loop)

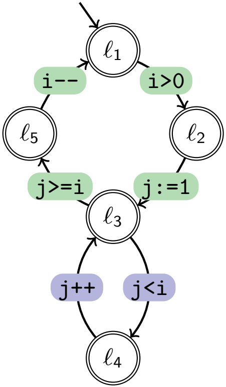
Function from program states to well-founded domain such that value is decreasing while executing the loop body.
Proof by contradiction for the absence of infinite executions.

Example: Bubble Sort

```
   program sort(int i, int a[])
ℓ1 while (i>0)
ℓ2      int j:=1
ℓ3      while(j<i)
             if (a[j]>a[i])
                  swap(a,i,j)
ℓ4           j++
ℓ5      i--
```

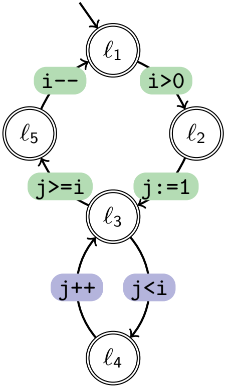
Example: Bubble Sort

```
program sort(int i)
ℓ1  while (i>0)
ℓ2      int j:=1
ℓ3      while(j<i)
ℓ4          j++
ℓ5      i--
```



Example: Bubble Sort

```
program sort(int i)
ℓ1 while (i>0)
ℓ2      int j:=1
ℓ3      while(j<i)
ℓ4           j++
ℓ5      i--
```

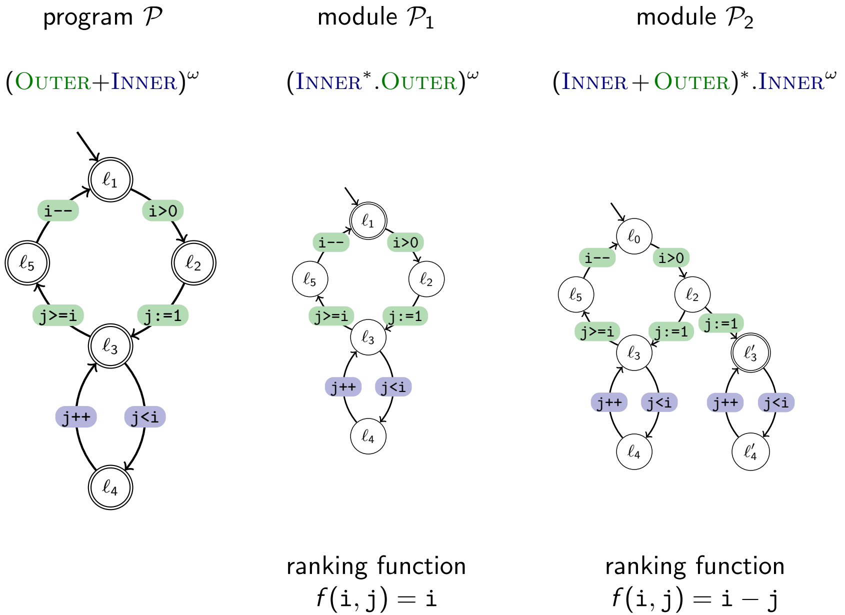


quadratic ranking function:

$$f(\mathtt{i},\mathtt{j}) = \mathtt{i}^2 - \mathtt{j}$$

lexicographic ranking function:

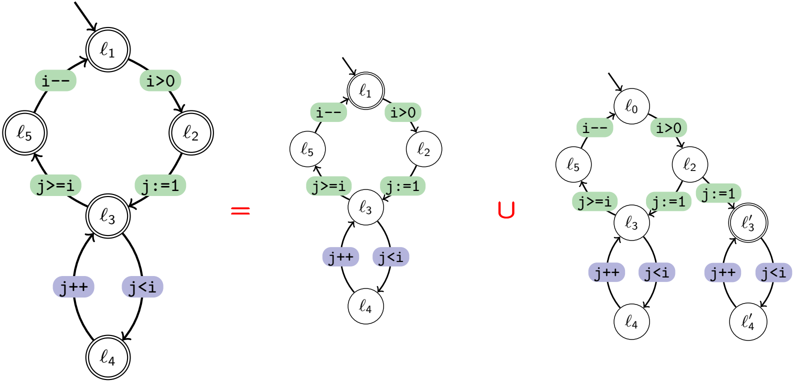
$$f(\mathtt{i},\mathtt{j}) = (\mathtt{i}, \mathtt{i} - \mathtt{j})$$



program $\mathcal{P}$
module $\mathcal{P}_1$
module $\mathcal{P}_2$
$(\textsc{Outer}+\textsc{Inner})^\omega$
$(\textsc{Inner}^*.\textsc{Outer})^\omega$
$(\textsc{Inner}+\textsc{Outer})^*.\textsc{Inner}^\omega$
ℓ_0
ℓ_1
ℓ_2
ℓ_3
ℓ_4
ℓ_5
ℓ_3'
ℓ_4'
i--
i>0
j>=i
j:=1
j++
j<i
ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i}$
ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i} - \mathtt{j}$

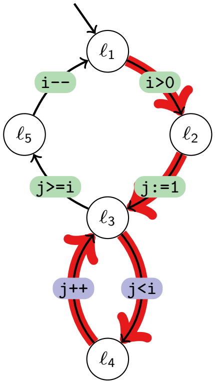
program $\mathcal{P}$ | module $\mathcal{P}_1$ | module $\mathcal{P}_2$

$$(\textsc{Outer}+\textsc{Inner})^\omega \;=\; (\textsc{Inner}^*.\textsc{Outer})^\omega \;+\; (\textsc{Inner}+\textsc{Outer})^*.\textsc{Inner}^\omega$$



ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i}$

ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i} - \mathtt{j}$



ℓ_1
i>0
ℓ_2
j:=1
ℓ_3
j<i
ℓ_4
j++
j>=i
ℓ_5
i--

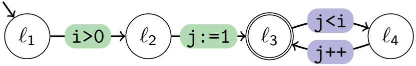
From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^{\omega}$,

From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

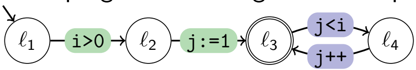
1. consider ω-trace as program with single while loop



From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

1. consider ω-trace as program with single while loop



2. synthesize ranking function

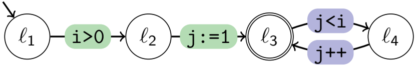
$$f(i,j) = i - j$$

Colón, Sipma **Synthesis of Linear Ranking Functions** (TACAS 2001)
Podelski, Rybalchenko **A complete method for the synthesis of linear ranking functions** (VMCAI 2004)
Bradley, Manna, Sipma **Termination Analysis of Integer Linear Loops** (CONCUR 2005)
Bradley, Manna, Sipma **Linear ranking with reachability** (CAV 2005)
Bradley, Manna, Sipma **The polyranking principle** (ICALP 2005)
Ben-Amram, Genaim **Ranking functions for linear-constraint loops** (POPL 2013)
H., Hoenicke, Leike, Podelski **Linear Ranking for Linear Lasso Programs** (ATVA 2013)
Cook, Kroening, Rümmer, Wintersteiger **Ranking function synthesis for bit-vector relations** (FMSD 2013)
Leike, H. **Ranking Templates for Linear Loops** (TACAS 2014)

From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

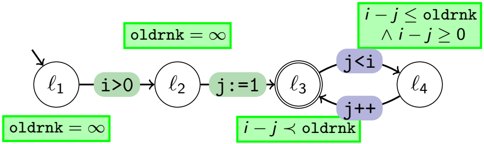
1. consider ω-trace as program with single while loop



2. synthesize ranking function

$$f(i,j) = i - j$$

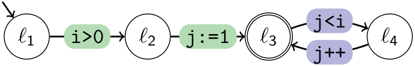
3. compute rank certificate



From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

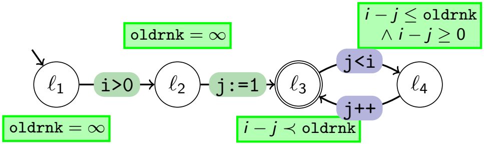
1. consider ω-trace as program with single while loop



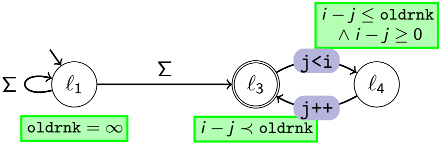
2. synthesize ranking function

$$f(i,j) = i - j$$

3. compute rank certificate



4. add additional transitions

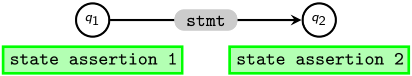


Generalization of Program with Rank Certificate

- Case 1: q_1 not accepting

Hoare triple { state assertion 1 } stmt { state assertion 2 }

automaton transition

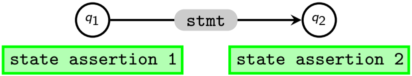


Generalization of Program with Rank Certificate

- Case 1: q_1 not accepting

  Hoare triple { `state assertion 1` } `stmt` { `state assertion 2` }

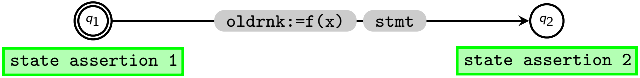
  automaton transition

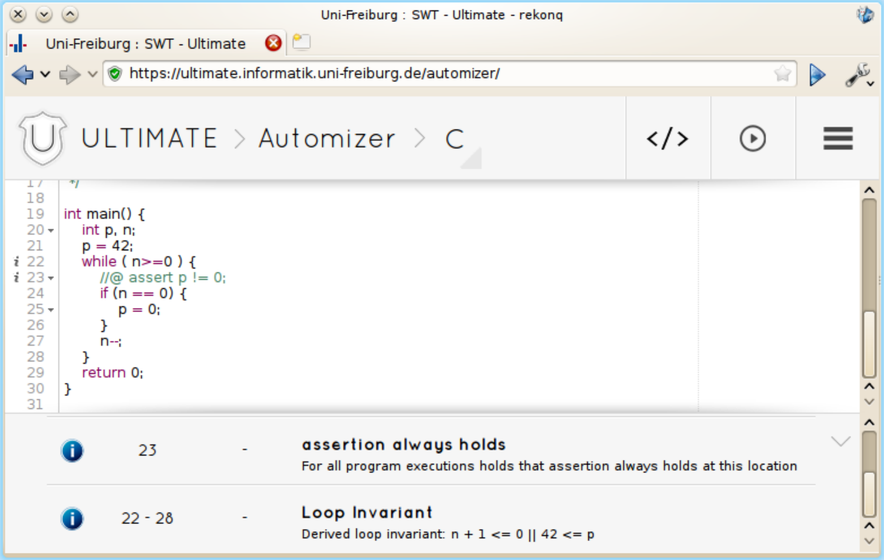
  

- Case 2: q_1 accepting

  Hoare triple { `state assertion 1` } `oldrnk:=f(x)` `stmt` { `state assertion 2` }

  automaton transition



http://ultimate.informatik.uni-freiburg.de/automizer

Thank you for your attention!

Bibliography

Matthias Heizmann.
Traces, interpolants, and automata: a new approach to automatic software verification.
PhD thesis, University of Freiburg, Freiburg im Breisgau, Germany, 2015.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski.
Software model checking for people who love automata.
In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2013.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski.
Termination analysis by learning terminating programs.
In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 797–813. Springer, 2014.